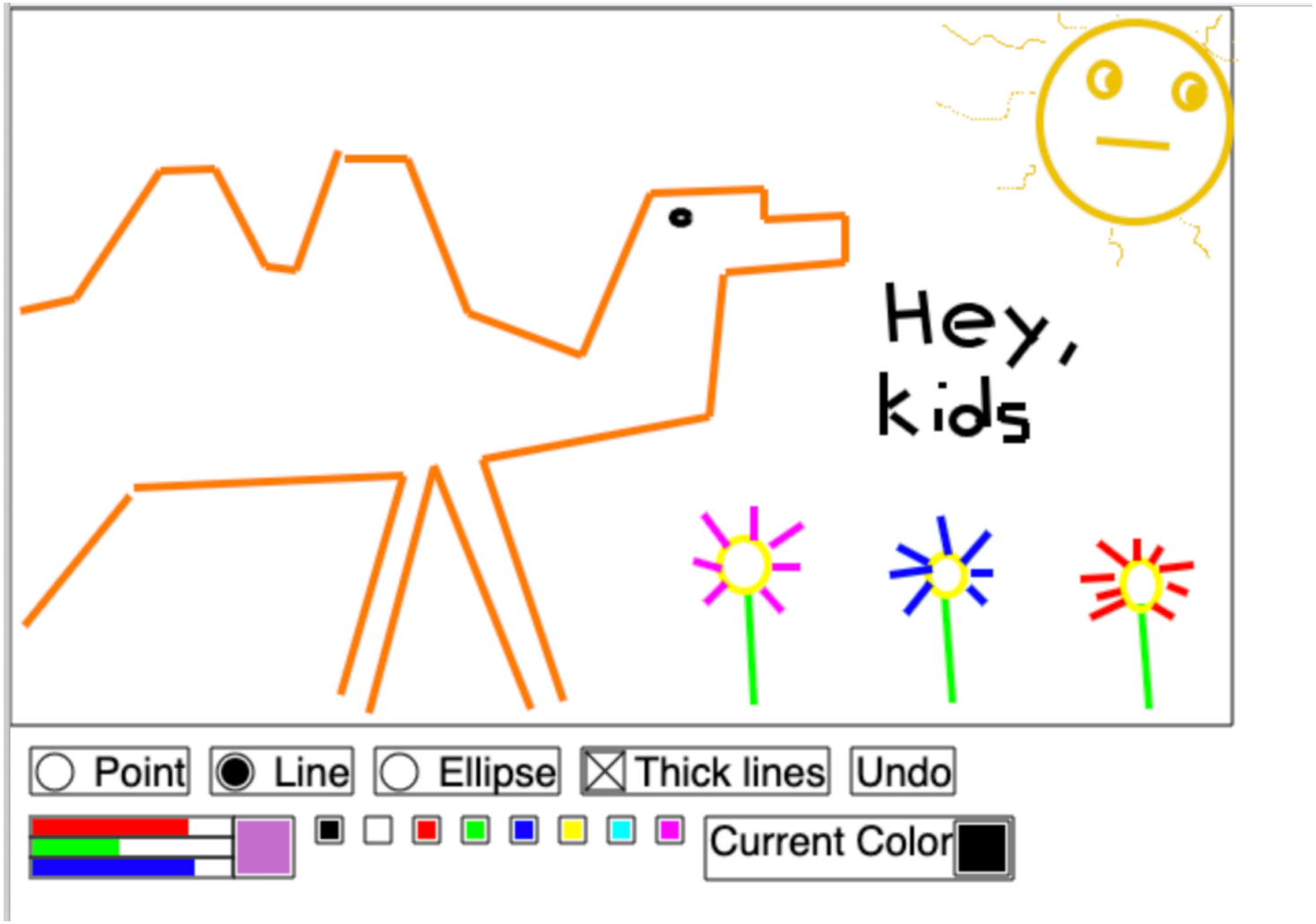
Programming Languages and Techniques (CIS120)

Lecture 19

GUI library: Simple Widgets

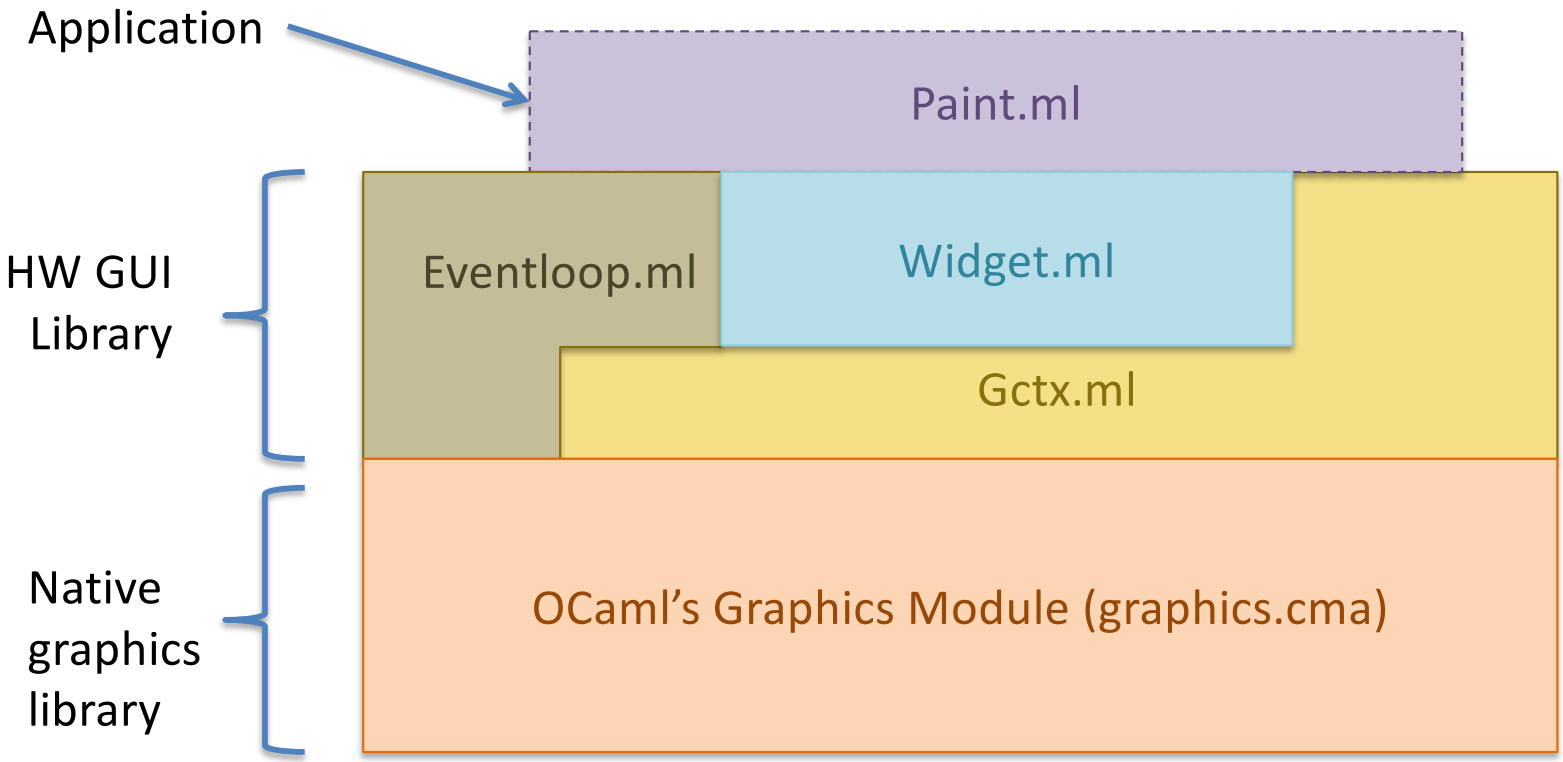
Chapter 18

Building a GUI library & application



Review: Widget Layout

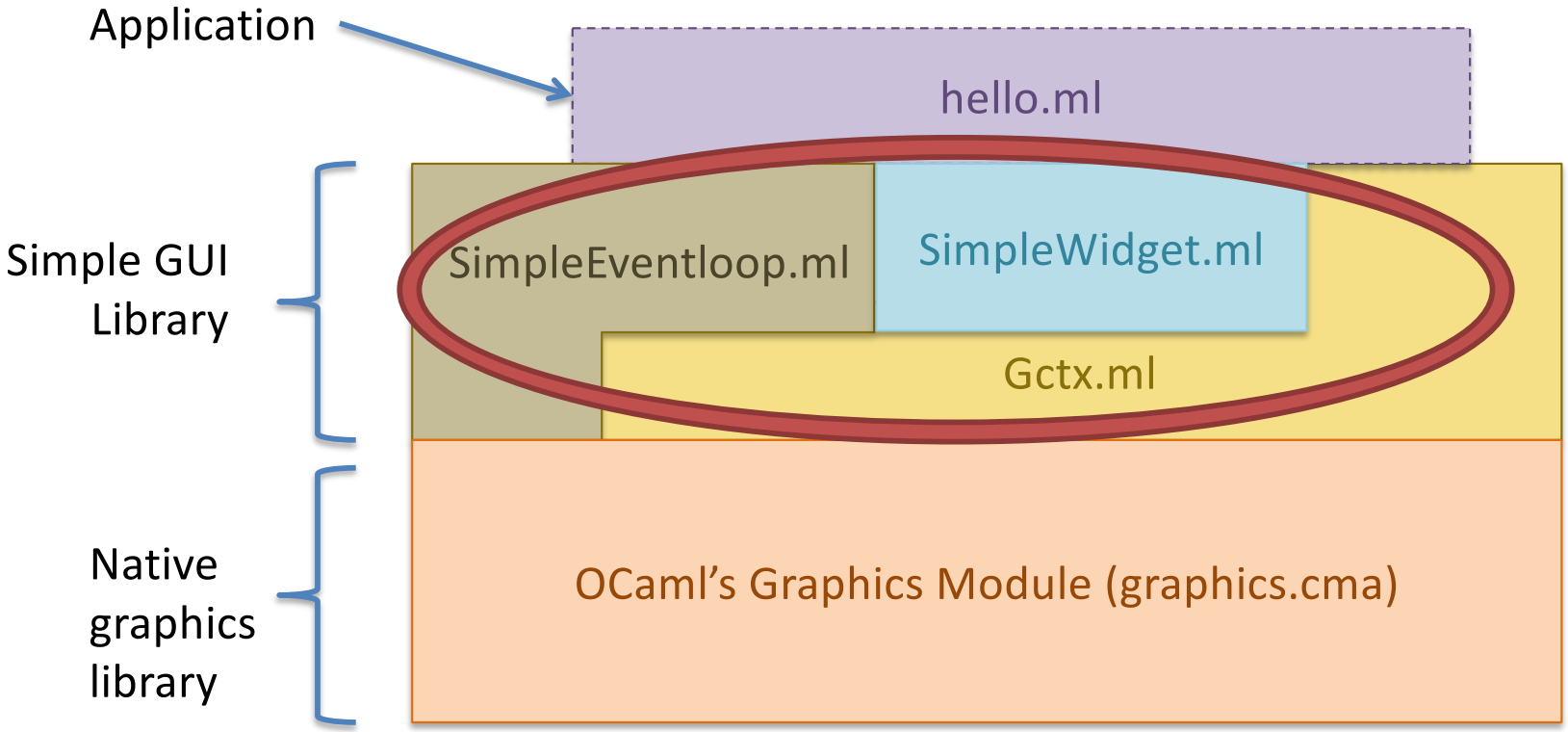
- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget’s current position



The graphics context isolates the widgets from the Graphics module.

Simplified Version

- Today: look at simplified version of widget and eventloop libraries.
- Simple applications (hello, fractalTree, swdemo)
- Project available via Codio (GUI Demo)



Review: Simple Widgets

simpleWidget.mli

```
(* An interface for simple GUI widgets *)
type widget = {
   repaint : Gctx.gctx -> unit;
   size    : unit -> (int * int)
}
val label  : string -> widget
val space  : int * int -> widget
val border : widget -> widget
val hpair  : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```

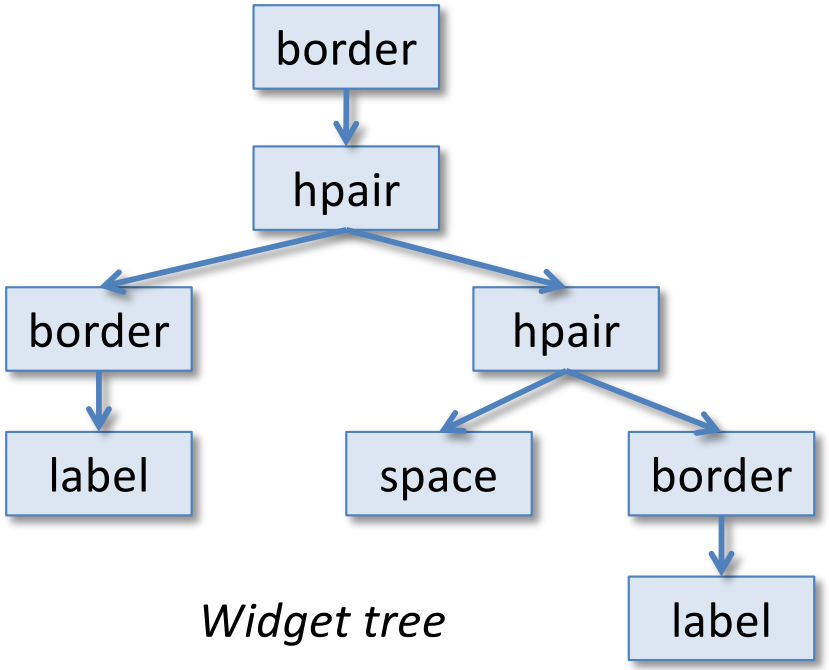
- You can ask a simple widget to repaint itself
- You can ask a simple widget to tell you its size

- Repainting is relative to a graphics context

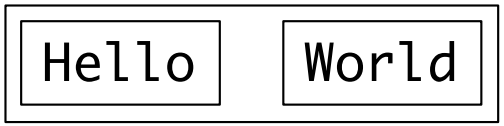
Review: A “Hello World” application

swdemo.ml

```
(* Create some simple label widgets *)
let l1 = label "Hello"
let l2 = label "World"
(* Compose them horizontally, adding some borders *)
let h =  border
           (hpair (border l1)
                  (hpair (space (10,10)) (border l2)))
```



Widget tree



On the screen

The canvas Widget

- Region of the screen that can be drawn upon
- Has a fixed width and height
- Parameterized by a repaint method
  - …which will directly use the Gctx drawing routines to draw on the canvas

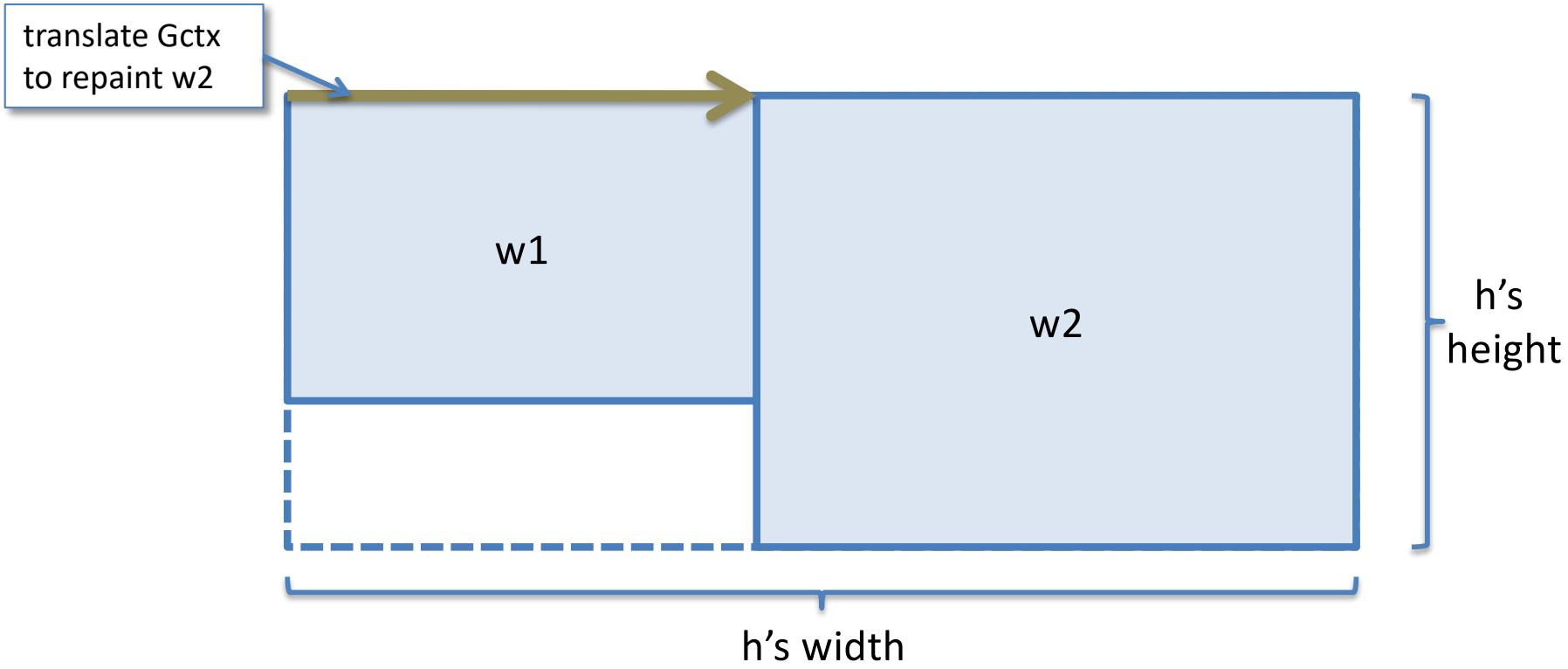
```
let canvas ((w,h):int*int) (r: Gctx.gctx -> unit) : widget =
{
  repaint = r;
  size = (fun () -> (w,h))
}
```

simpleWidget.ml

Nested Widgets

Containers and Composition

The hpair Widget Container



- `let h = hpair w1 w2`
- Creates a horizontally adjacent pair of widgets
- Aligns them by their top edges
  - Must translate the Gctx when repainting w2
- Size is the *sum* of their widths and *max* of their heights

The hpair Widget

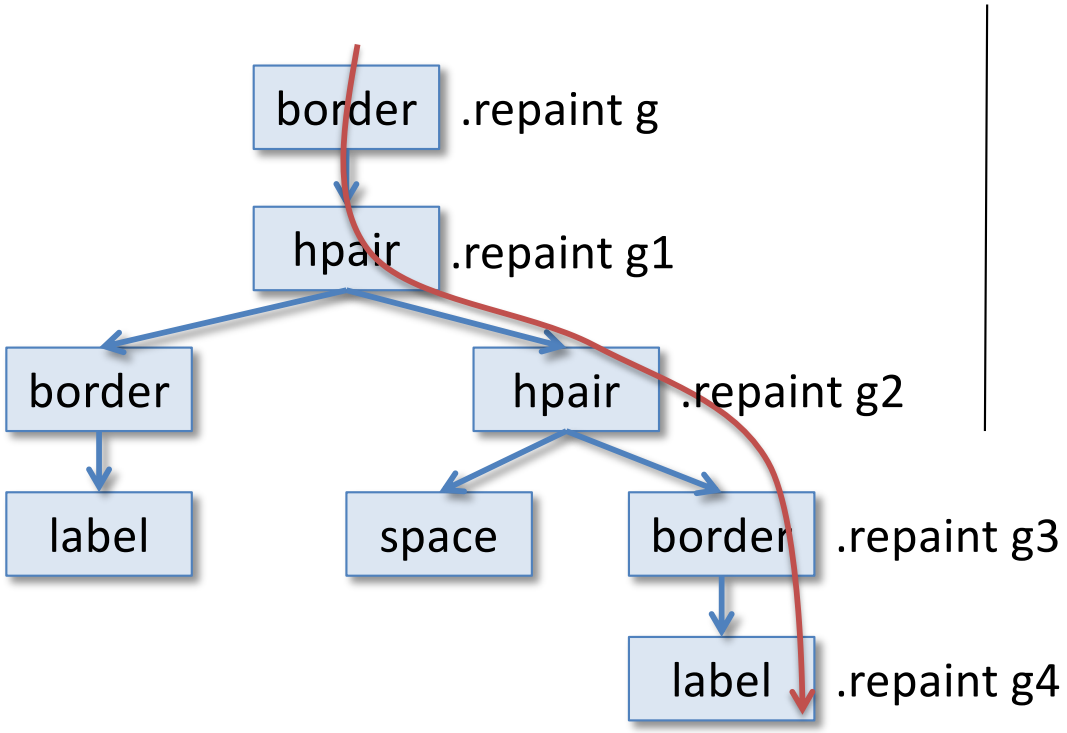
simpleWidget.ml

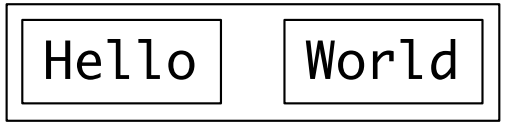
```
let hpair (w1: widget) (w2: widget) : widget =
  {
    repaint = (fun (g: Gctx.gctx) ->
            let (x1, _) = w1.size () in begin
              w1.repaint g;
              w2.repaint (Gctx.translate g (x1,0))
              (* Note translation of the Gctx *)
            end);

    size = (fun () ->
            let (x1, y1) = w1.size () in
            let (x2, y2) = w2.size () in
            (x1 + x2, max y1 y2))
  }
```

Translate the Gctx to shift w2's position relative to widget-local origin.

Drawing: Containers

Container widgets propagate repaint commands to their children:



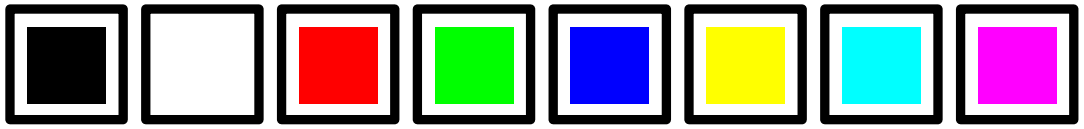


Widget tree

```
g1 = Gctx.translate g (2,2)
g2 = Gctx.translate g1 (hello_width,0)
g3 = Gctx.translate g2 (space_width,0)
g4 = Gctx.translate g3 (2,2)
```

On the screen

Container Widgets for layout

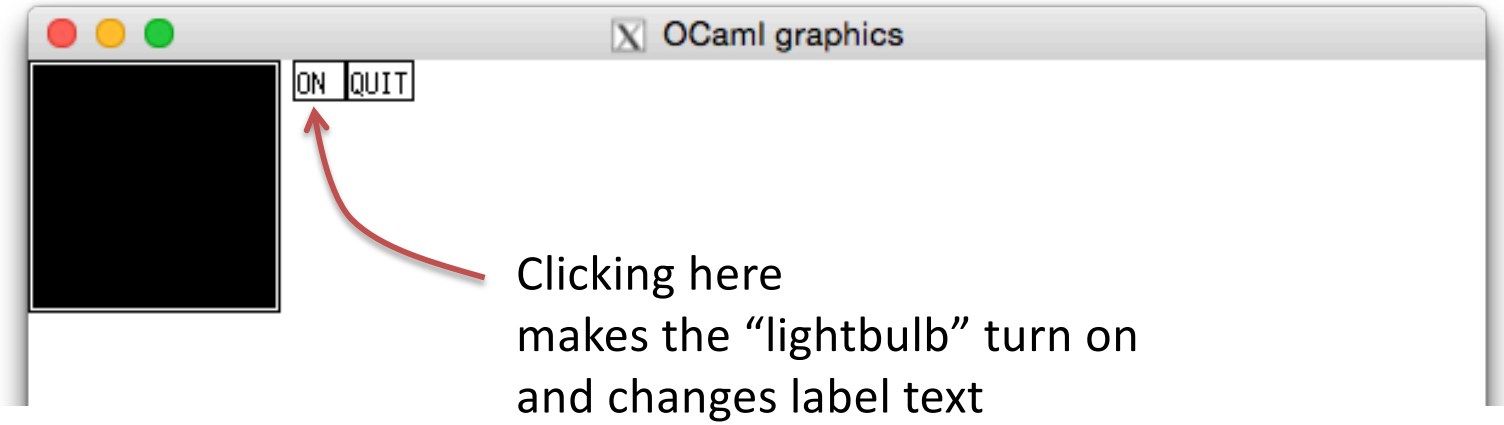


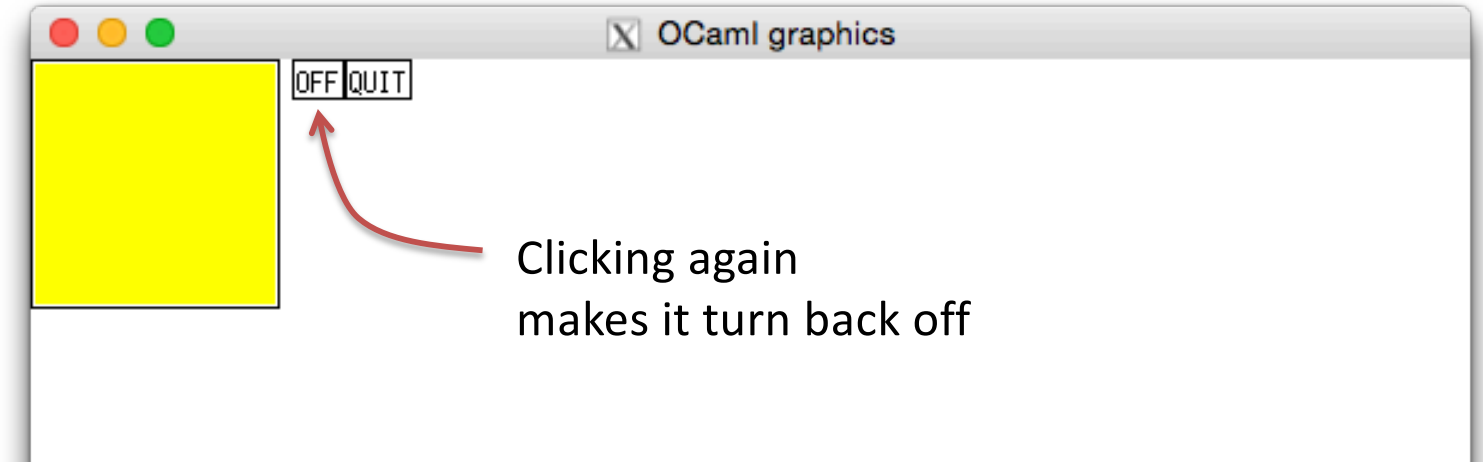
```
let color_toolbar : widget = hlist
   [ color_button black;  spacer;
     color_button white;  spacer;
     color_button red;    spacer;
     color_button green;  spacer;
     color_button blue;   spacer;
     color_button yellow; spacer;
     color_button cyan;   spacer;
     color_button magenta]
```

paint.ml

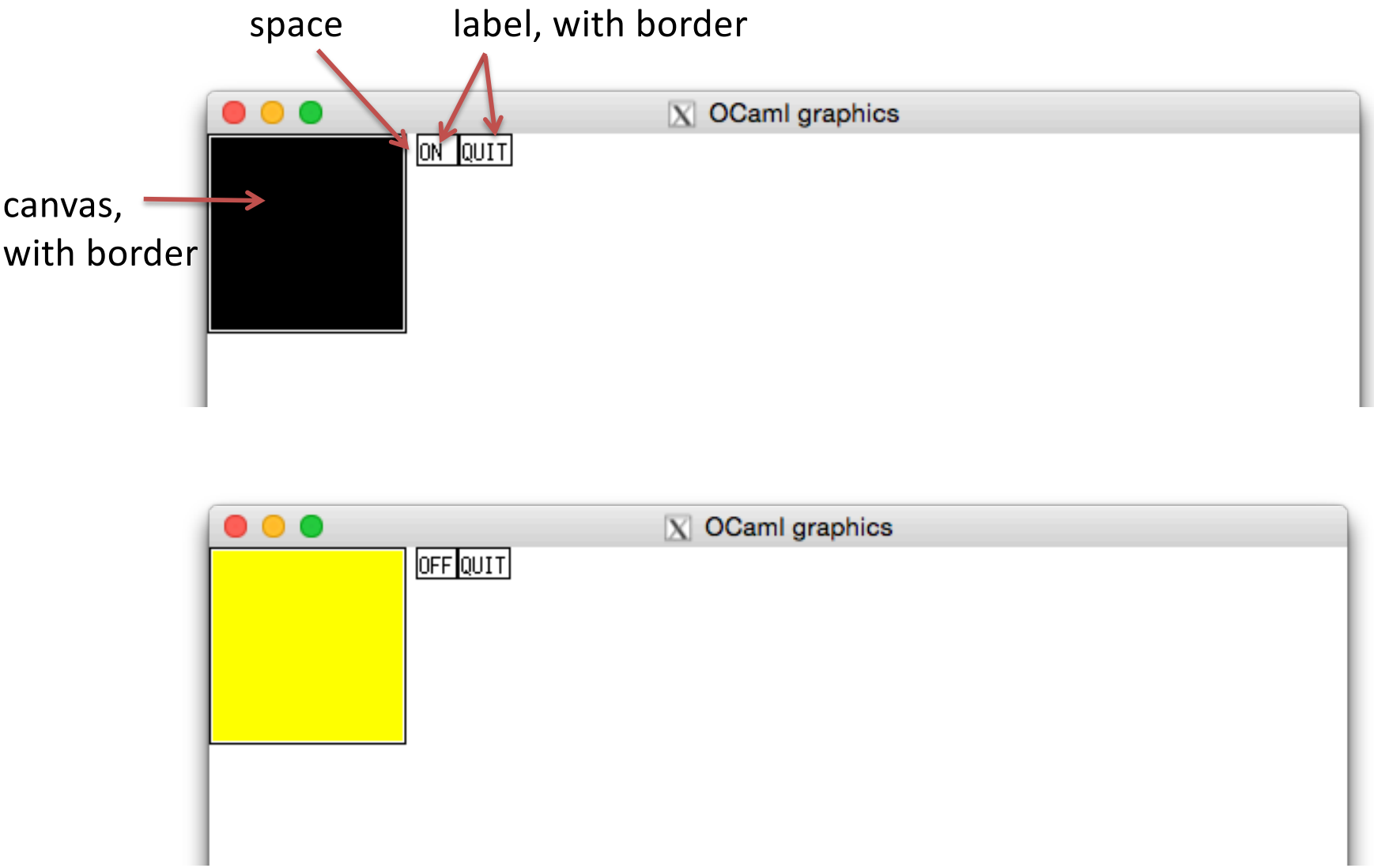
hlist is a container widget. It takes a list of widgets and turns them into a single one by laying them out horizontally (using hpair).

lightbulb demo





lightbulb demo



Events and Event Handling

Event loop with event handling

```
let run (w:widget) : unit =
  let g = Gctx.top_level  in      …create the initial gctx…
  Graphics.loop                   …wait for user input
    (fun e ->
       clear_graph ();
       w.handle g e;              …inform widget about the event…
       w.repaint g)               …update the widget's appearance…
```

Eventloop

```
let rec loop (f: event -> unit) : unit =
  let e = wait_next_event () in
  f e;
  loop f
```

Graphics

Events

gcxt.mli

```
type event

val wait_for_event : unit -> event

type event_type =
  | KeyPress of char  (* User pressed a key                    *)
  | MouseDown         (* Mouse Button pressed, no movement     *)
  | MouseUp           (* Mouse button released, no movement    *)
  | MouseMove         (* Mouse moved with button up            *)
  | MouseDrag         (* Mouse moved with button down          *)

val event_type : event -> event_type
val event_pos  : event -> gctx -> position
```

Remember:

The graphics context translates the location of the event to widget-local coordinates

Reactive Widgets

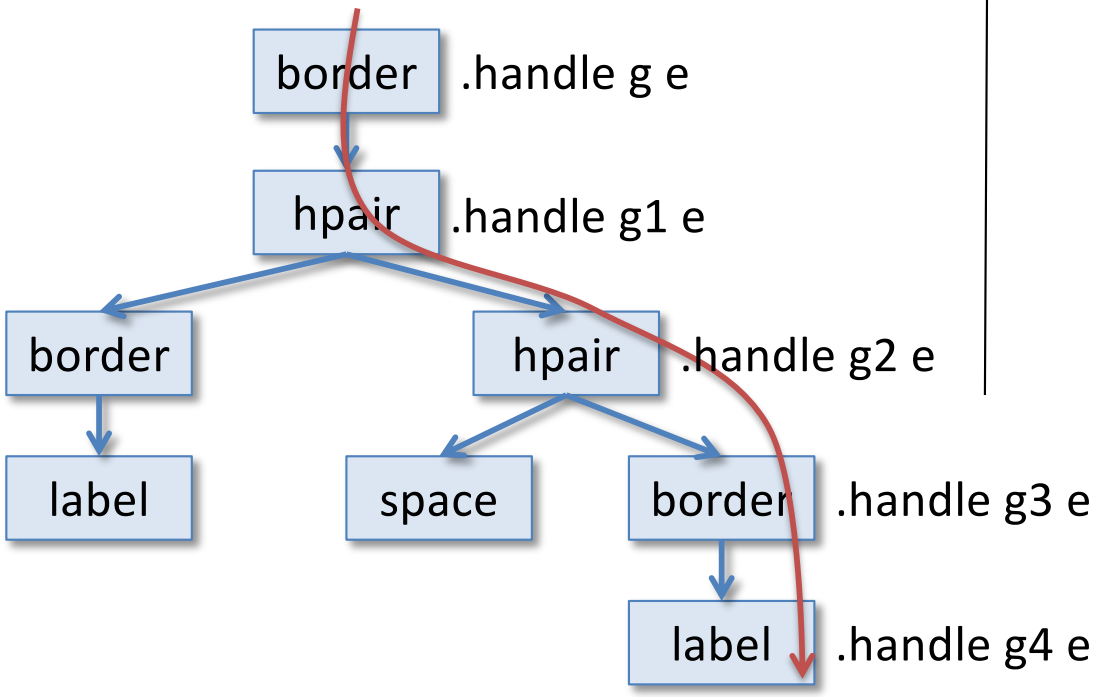
widget.mli

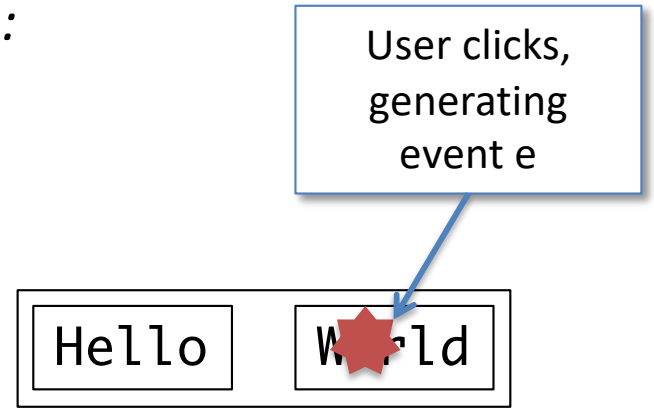
```
type widget = {
  repaint : Gctx.gctx -> unit;
  size    : unit -> Gctx.dimension;
  handle  : Gctx.gctx -> Gctx.event -> unit
}
```

- Widgets now have a “method” for handling events
  - The eventloop waits for an event and then gives it to the root widget
  - The widgets forward the event down the tree, according to the position of the event

Event-handling: Containers

Container widgets propagate events to their children:





Widget tree

On the screen

Routing events through container widgets

Event Handling: Routing

- When a container widget handles an event, it passes the event to the appropriate child
- The Gctx.gctx must be translated so that the child can interpret the event in its own local coordinates.

widget.ml

```
let border (w:widget):widget =
  { repaint = …;
    size = …;
    handle = (fun (g:Gctx.gctx) (e:Gctx.event) ->
       w.handle (Gctx.translate g (2,2)) e);
  }
```

Consider routing an event through an hpair widget constructed by:

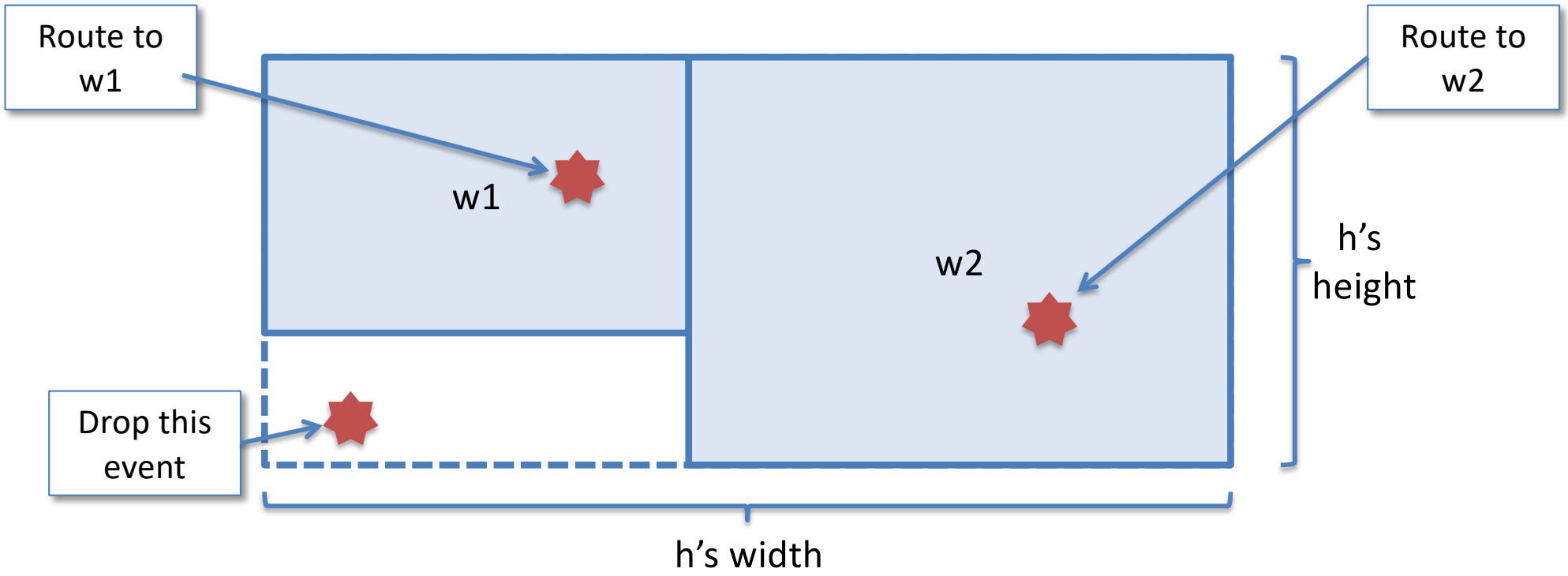
```
let hp = hpair w1 w2
```

The event will always be propagated either to w1 or w2.

1. True
2. False

Answer: False

Routing events through hpair widgets



- There are three cases for routing in an hpair.
- An event in the "empty area" should not be sent to either w1 or w2.

Routing events through hpair widgets

- The event handler of an hpair must check to see whether the event should be handled by the left or right widget.
  - Check the event's coordinates against the *size* of the left widget
  - If the event is within the left widget, let it handle the event
  - Otherwise check the event's coordinates against the right child's
  - If the right child gets the event, don't forget to translate its coordinates

```
handle =
 (fun (g:Gctx.gctx) (e:Gctx.event) ->
   if event_within g e (w1.size ())
   then w1.handle g e
   else
   let g = (Gctx.translate g (fst (w1.size ()), 0)) in
     if event_within g e (w2.size ())
     then w2.handle g e
     else ())
```

Stateful Widgets

How can widgets react to events?

(not very useful first stab at a)

A stateful `label` Widget

```
let label (s: string) : widget =
  let r = { contents = s } in
  { repaint = (fun (g: Gctx.gctx) ->
                  Gctx.draw_string g (0,0) r.contents);
    handle  = (fun _ _ -> ());
    size    = (fun () -> Gctx.text_size r.contents)
  }
```

- The label object can make its string mutable. The “methods” can refer to this mutable string.
- But how can we change this string in response to an event?

A stateful `label` Widget

widget.ml

```
type label_controller = { set_label: string -> unit }

let label (s: string) : widget * label_controller =
  let r = { contents = s } in
    ({ repaint = (fun (g: Gctx.gctx) ->
                  Gctx.draw_string g (0,0) r.contents);
       handle  = (fun _ _ -> ());
       size    = (fun () -> Gctx.text_size r.contents)
     }
    ,
    { set_label = fun (s: string) -> r.contents <- s })
```

- A *controller* gives access to the shared state.
  - Here, the `label_controller` object returned by `label` provides a way to set the label string